Dissecting Java Server Faces for Penetration Testing

Aditya K Sood (Cigital Labs) & Krishna Raja (Security Compass)

Version 0.1

August 25, 2011

Abstract

This paper sheds light on the findings of security testing of Java Server Faces (JSF). JSF has been widely used as an open source web framework for developing efficient applications using J2EE. JSF is compared with ASP.NET framework to unearth potential security flaws.

Contents

1 Acknowledgments

We would like to thank couple of our friends and security researchers who helped us in shaping this paper.

- Giorge Maone (NoScript)
- Juliano Rizzo (Netifera)

In addition, we would also like to thank Gary McGraw for providing useful insight into the paper. A sincere gratitude to all the researchers who are engaged in constructive research for the security community. Lastly, we sincerely want to thank our security teams at SecNiche Security Labs and Security Compass respectively for supporting us in doing security research.

2 Overview

In present times, software security has become an indispensable part of software development life cycle. The penetration testing approach varies with respect to web development frameworks and platforms. With the advent of advanced level of attacks, it has become crucial to raise the standards of penetration testing. An aggressive security testing approach is required to detect the inherent vulnerabilities and to develop robust security solutions in order to thwart sophisticated attacks. Owing to the seamless pace of security research, a plethora of vulnerabilities are being unearthed in web frameworks and software. Thus, for effective penetration testing, the security model and web framework architecture should be dissected appropriately.

OWASP has been used widely as the de facto standard of penetration testing of web applications and frameworks with its Top 10 attack vectors. However, the penetration testing methodology should not be constrained to this standard and must cover the advanced set of attack vectors that should be tested to validate the strength of web frameworks.

This paper is divided into two parts. In the first part, we discuss the internals of JSF, a Java based web application framework and its inherent security model. In the second part, we discuss about the security weaknesses and applied security features in the JSF. In addition, we also raise a flag on the security issues present in JSF in order to conduct effective penetration testing.

3 Inside JSF Framework

Java Server Faces (JSF) is an industry standard and a framework for building component-based user interfaces for web applications. JSF has certain standards and is implemented using Reference Implementation (RI) by Sun Microsystems, Apache MyFaces and Oracles ADF Faces. JSF primarily consists of Java Beans, Event Driven development and JSF component tree.

With the advent of JSF, the control has been handed over to the developers for implementing security features such as authorization. As a result of this change, it has become more crucial for the developers to understand the implementation of security controls in JSF framework. A good security design practice requires that authorization (security controls) should be handled at a central location (Servlet Filter associated with the application front controller). JSF has built-in extension points provided by the JSF architecture. As JSF has no proprietary security architecture, the security has to be imposed in a customized fashion. This is usually done in two ways

- The developer can design a custom ViewHandler that adds security checks for createView and restoreView. However, it is not considered as the best security practice because there is no guarantee that the custom security ViewHandler is executed before the default ViewHandler. This leads to security exceptions while handling requests from the web clients
- The developer can design a phaseListener that adds security definitions to restoreView and invokeAction phases. This can be implemented in JSF faces-config.xml file or the developer can do it dynamically by writing a customPhase listener.

3.1 JSF Security Architecture

JSF is used for designing web based rich internet applications over the J2EE framework. Java applications are mostly designed using Model, View, and Controller (MVC) architecture due to the need for real time deployment. J2EE security can be implemented through Java Authentication and Authorization Service (JAAS) and container-managed security. JAAS implements fine-grained access control through external Java permission classes, which provides user with a list of resources and allowed actions. Before J2EE, the security controls were implemented within the logic itself. J2EE framework has a declarative security mechanism in which security controls are applied through web.xml deployment descriptors which in turn are handled by the J2EE container at runtime. In container managed security, controls are applied using authorization which is explicitly enforced on the URL patterns (requests that are issued by the web client).

Generally, the controller is responsible for implementing security controls whereas the view and model part are used for hiding information and applying logic based on the access roles of the user respectively. However, a good design

practice suggests that the security should be implemented over all three layers. as presented in figure 1.

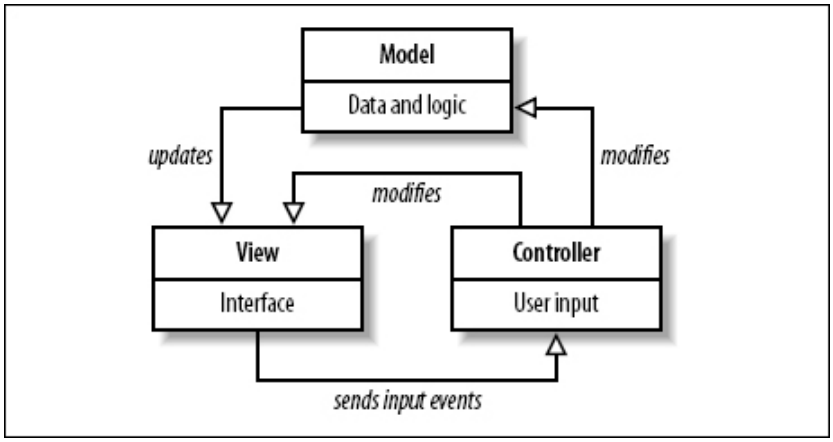

Figure 1: MVC - Model View and Controller Architecture

JSF has implemented the concept of validators which can be used to verify user input at a view level and model level.

3.1.1 JSF Faces-Config.xml and Web.xml

The rich life cycle of various individual phases are explicitly specified in the faces-config.xml file. Some of the events included in the file are Restore View, Apply Request Values, Process Validation, Update Model Values, Invoke Application, and Render Response. Developers should always consider the fact that faces-config.xml file has no mention of security and all security constraints must be specified in web.xml file.

4 Penetration Testing JSF Framework

In this section, we are going to present the variation in the applied security model in JSF frameworks, security weaknesses and the right way to test them.

4.1 JSF ViewState Anatomy

JSF uses ViewState functionality as similar to ASP.NET. However, there are certain differences in the way JSF and ASP.NET handle ViewState. Generally, as we all know, the ViewState parameter is used to maintain the state of HTTP request specific to a web page. This functionality proves beneficial, but it requires appropriate implementation in the deployed environment. It has been noticed that ViewState analysis of ASP.NET and JSF is misunderstood.

4.1.1 Differential Behavior - ViewState in ASP.NET and JSF

There are number of differences in ViewState implementation in JSF and ASP.NET which should be taken into consideration while performing analysis of the ViewState. These are discussed as follows

- JSF does not encrypt the ViewState parameters by default. JSF works on the concept of serialization and compression. In general scenarios, once the information is serialized, it is compressed using the GZIP algorithm before being pushed onto a base-64 encoder. Mojarra displays this behavior, whereas latest versions of Apache My faces perform encryption by default but MAC is not enabled (prone to padding oracle attack).
- Compression plays a critical role in optimizing requests in JSF and this is primarily implemented through the *"com.sun.faces.compressViewState"* context parameter. JSF also uses the *"com.sun.faces.compressJavaScript"* context parameter to remove the whitespaces in rendering JavaScript. However, this parameter does not have much impact on security testing.
- In Apache JSF and Mojarra (SUNs RI), the encryption of ViewState is only possible through explicit declaration in the Java Naming and Declaration Interface (JNDI) using a password string as presented in listing 1.

```
<env-entry>
 <env-entry-name>com.sun.faces.ClientStateSavingPassword</env-
     entry-name>
 <env-entry-type>java.lang.String</env-entry-type>
 <env-entry-value>[Provide Random Value]</env-entry-value>
</env-entry>
```

Listing 1: Implementing Encryption using JNDI

- In ASP.NET applications, session state is enabled by default which requires session cookies to navigate through browser sessions, which is well

understood. In ASP.Net, the *"ViewStateEncryptionMode.Auto"* mode is set by default which decides whether a specific web page has to have an encrypted ViewState or not in order to reduce the processing load. However, it is always advised to encrypt the full ViewState in every webpage by declaring the *"<%@Page ViewStateEncryptionMode="Always" %>"* property. This ensures that ViewState data could not be retrieved.

- In ASP.NET, Message Authentication Code (MAC) is also computed by default and appended in the base 64 encoding in order to avoid the tampering of ViewState with arbitrary data. The biggest problem in implementing MAC is that it has to be explicitly specified on the webpage with the *"enabledViewStateMac"* parameter to be true otherwise MAC is not enabled by default. It is advised that the MAC key should be larger in size in order to reduce the attack surface. Usually, the GUID of the machine is used as a MAC key.

Some of the generic ViewState decoders which fail in JSF may work fine in ASP.NET ViewState decoding. ViewState decoder designed by plural-sight [2] fails for JSF and works fine for ASP.NET as it only works in .NET environment. Figure 2 shows that tool raises red alert while handling JSF ViewState and should not be used for the analysis of JSF ViewState.

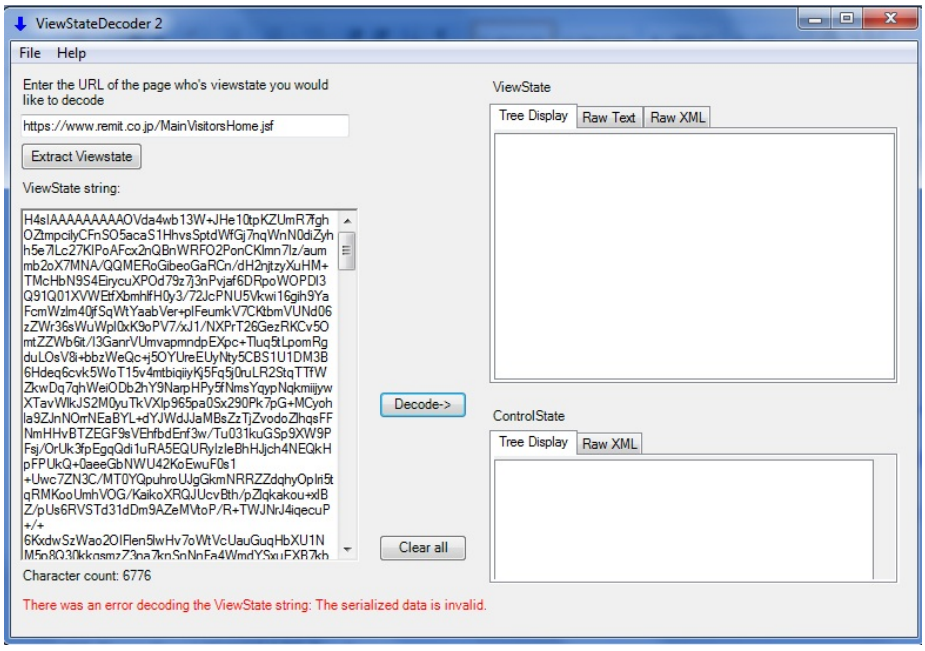

Figure 2: ViewState Decoder Fails for JSF

Netifera group has also released a tool termed as POET [3] which should be used for testing ViewState in JSF. Figure 2 shows the successful decoding of ViewState in JSF. However, some of the data is gzipped which can be further unzipped fully. Even this information raises an alert about the insecure implementation of ViewState in JSF.

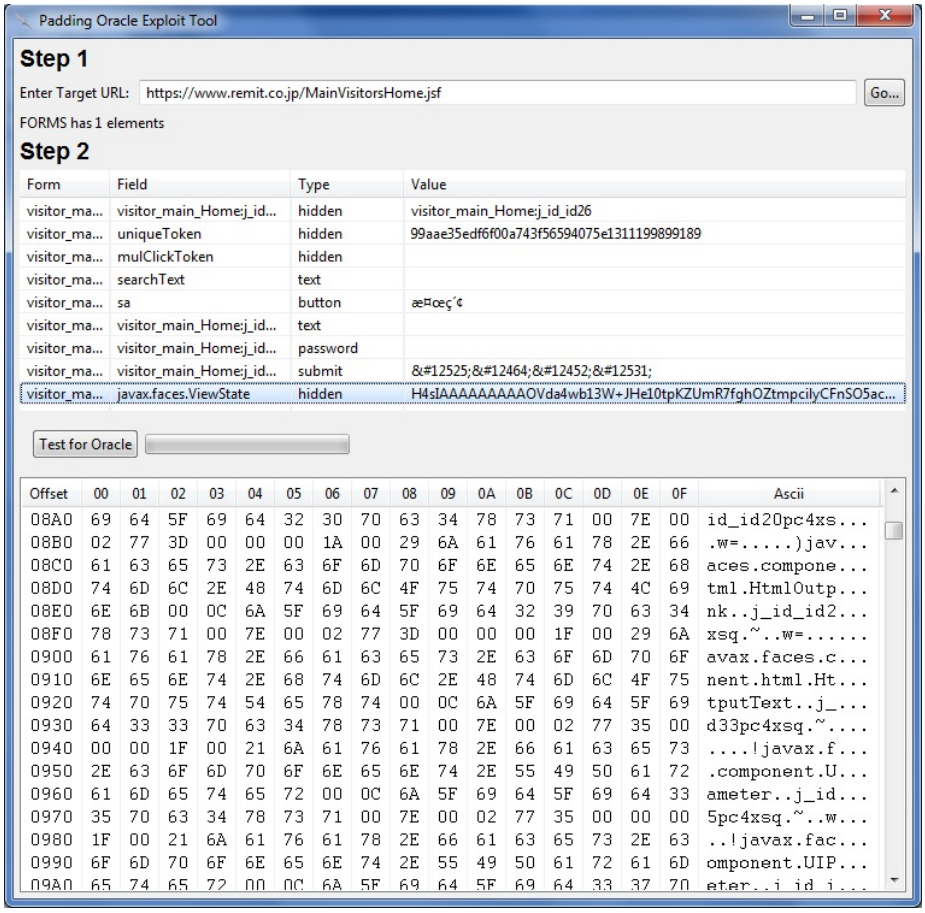

Figure 3: Successful Decoding of ViewState

One can also use the Deface [4],[5] tool for testing ViewState in JSF which is released by SpiderLabs for aggressive testing of JSF framework.

4.2 Scrutinizing Padding - Testing Oracle

With the advent of new technologies, more sophisticated attack patterns are being noticed in the routine life. Last year, the discovery of padding oracle attacks [6] has dismantled the implementation of encryption in web frameworks.

Due to the padding problem, it is possible to decrypt the ViewState effectively. In certain versions of ASP.NET, it is possible to download the web.config file on the local machine by decrypting the content of files using padding oracle attacks. This is implemented by exploiting the encrypted string that is passed to ScriptResource.axd and WebResource.axd and padding it appropriately. Microsoft released patches for the insecure cryptographic implementation in ASP.NET due to padding oracle attacks [10]. It was noticed that some of the applied patches were not correct and robust. Figure 3 shows how exactly the robustness of applied patch can be verified.

As demonstrated at IEEE Security Symposium this year, it is possible to build rogue requests using padding oracle which can be further exploited to query sensitive information from the web server. This has proved the fact that erroneous implementation of cryptography [7] can seriously dismantle the system and the web is no exception.

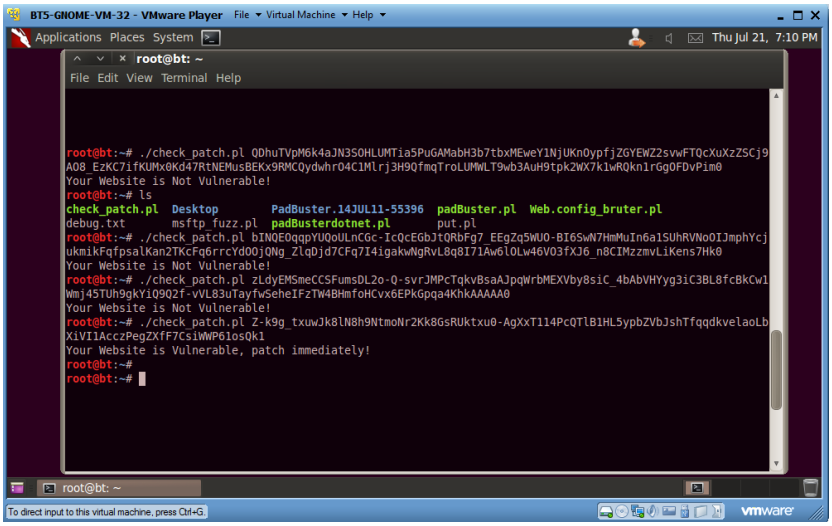

Figure 4: Checking Validity of Applied Patch [WebResource.axd/ ScriptResource.axd]

This can be successfully done through Padbuster [8] tool in ASP.NET. However, this tool and its variant successfully work for vulnerable versions of ASP.NET, provided insecure encryption is applied. This tool has also been added in the latest version of Backtrack penetration testing framework.

The padding oracle attacks can be successfully conducted in JSF. The first step is that encryption is not applied in ViewState by default (Mojarra frameworks). Even if encryption is applied in certain deployed JSF frameworks (Apache MyFaces) , the integrity is not protected using MAC as discussed earlier by default. This is the most frivolous flaw that is impacting JSF at

a large scale. A number of websites using JSF in a real time environment are still vulnerable and are running in default insecure state. The ViewState encryption strength in JSF can be checked using POET as discussed. The tool verifies whether the encryption is applied or not. If it is applied, then it has an inbuilt module to decrypt the ViewState using oracle padding. The tool follows the concept of tampering a single byte in the encrypted JSF ViewState (last block) to verify whether ViewState padding is done appropriately or not based on HTTP error fingerprinting. JSF usually ignores the inserted block during serialization which helps the tool to go on decrypting the ViewState without any hassles. The practical usage of tool can be seen here [9].

4.2.1 Experiment - Fuzzing Oracle

We conducted a number of tests on one of the vulnerable websites to show the impacts of the padding oracle. The tests are based on manual fuzzing. The aim is to present the variation in the error responses when encrypted ViewState is tampered. One thing that should be taken into account while performing this test is that ViewState has to be encrypted. The test should not be executed against ViewState that is compressed using GZIP. In addition, the ViewState should be fuzzed using multiples of 8 because the block size that is used in CBC encryption has a similar technique. The nature of a response to a padded buffer varies between applications.

Step 1: Injecting random buffer in ViewState as a multiple of 8. Figure 5 shows how the application reacts.

Step 2: At this point, we got a crypto padding error in step 2, on continuous playing around with padding in ViewState; we received different error as presented in figure 6.

Considering this scenario, one can continue fuzzing the request, until it is accepted by the application. There is a typical way of doing padding in CBC and that can be used in all scenarios as discussed here [11]. One can opt for various methods to pad CBC encryption.

4.3 JSF Anti CSRF - Truth Behind the Scenes

In reality, JSF does not have an aggressive built-in CSRF protection. Anti CSRF support is required for protection against Cross Site Request Forging (CSRF) attacks. However, the implementation of anti CSRF depends a lot on the design of he in the required framework. ViewState is used for preserving the state of web pages and can be used in conjunction with another configuration parameters to prevent CSRF attacks. However, one can perform certain logic tests to initially detect whether the application is vulnerable to CSRF attacks or not.

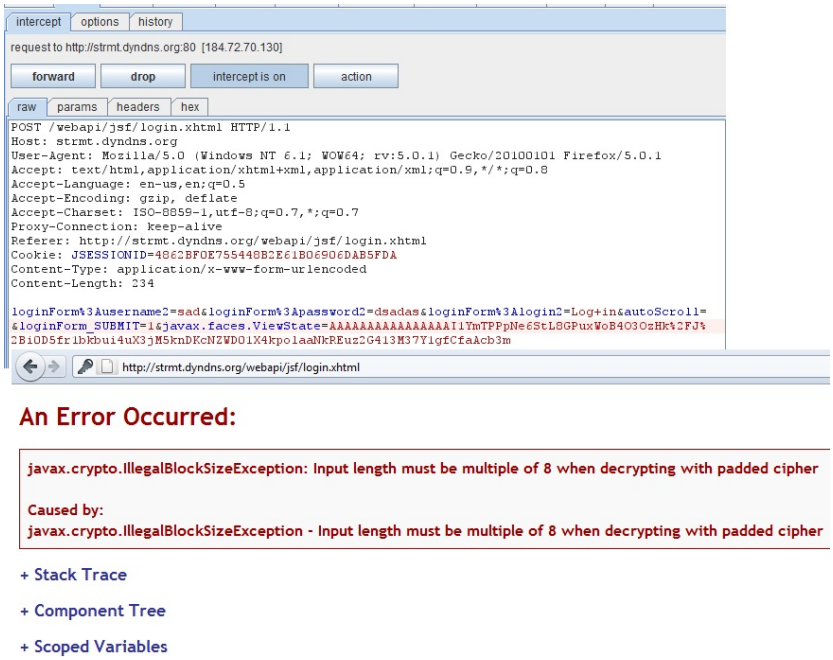

Figure 5: Fuzzing Request / Error in Block Size

Generally, if the ViewState is implemented on the server side, then it is a good security practice that the application should send a ViewState ID token as a hidden element in the HTML tag so that it can accompany legitimate requests from the client side. If the application is only implementing ViewState on the server side and is not using any ViewState ID, then it is possible that CSRF is not handled appropriately. Tokens generated by using *"javax.faces.ViewState"* (sequential) are easy to guess if not encrypted properly.

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.
    ViewState" value="j_id2"/>
```

Listing 2: Implementing ViewState Tracking on Server Side

As presented in listing 2, the j_id2 parameter is set for the ViewState tracking on the server side. The attacker designs the next request in that session with ViewState id as j_id3, j_id4 and so on which will be treated as legitimate by the server. In Apache MyFaces *"org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION"* has a default value of 20 where as IBM Web sphere *"com.sun.faces.numberOfViewsInSession"* has 15. These parameters specify the number of views that are stored in the session when server side state saving is used.

NOTE: In JSF, it is considered that ViewState can be used to prevent CSRF

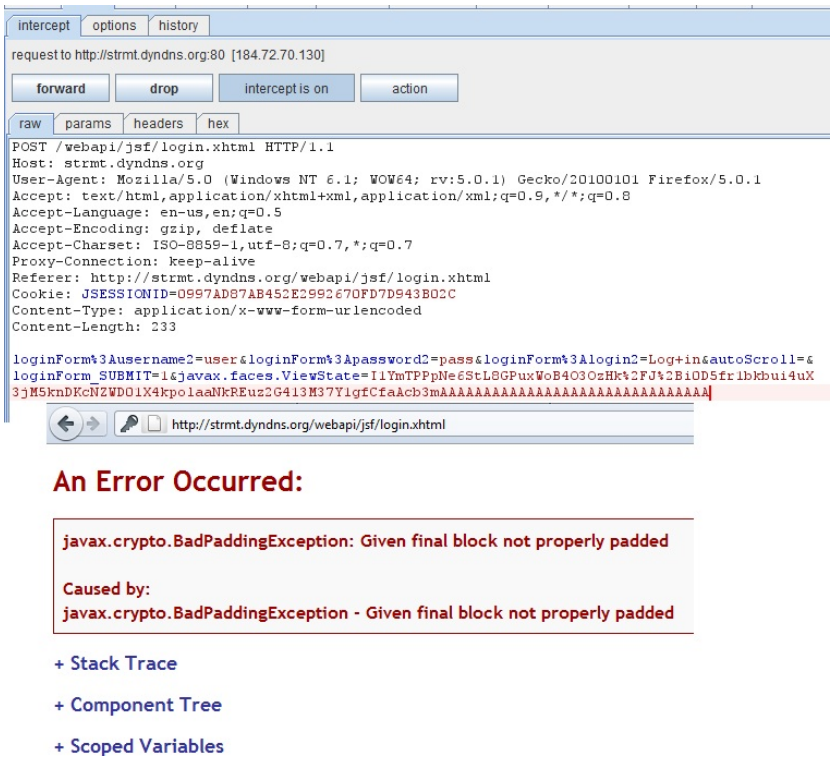

Figure 6: Fuzzing Request / Error in Last Block

attacks when collaboratively used with the JSESSIONID. As we have been discussing, ViewState implementation matters a lot. Now it has become possible to re encrypt the tampered ViewState and deliver it back to the server. Encrypting ViewState and sending data over HTTPS are not the protection mechanisms against CSRF attacks. This has been widely misunderstood in the developer community.

4.3.1 Implementing CSRF Protection - The Right Way

Strong CSRF implementation in JSF can be implemented as

- Applying Anti CSRF filters such as *"org.apache.catalina.filters.CsrfPreventionFilter"*. The inbuilt class uses the *"java.util.Random"* if explicitly specified by the developer otherwise *"java.security.SecureRandom"* will be used by default. One can also use OWASP CSRF Guard to integrate third party filters into JSF.

- If the ViewState session Id is to be used with every request then it must be strongly encrypted and an appropriate MAC should be applied in order

to preserve integrity.

- It is also possible to design custom CSRF filters with strong functions that generate random tokens. This is possible by creating a CSRF Session listener class that overrides every request with HTTP listener class and appends a random token in every request for a particular session. There is also a possibility of adding <s: token> an element in <h:form> the tag that automatically initiates the CSRF protection. Framework that supports <s: token> are Apache Shale, MyFaces and JBOSS Seam.

- The real world examples will look like as presented in listing 3

```
<input type="hidden" name="j_idt7:j_idt7_CSRFToken" value="0
    c776040ff77d3af5acce4d4c59a51411eb960bd" />
```

Listing 3: Implementing CSRF Tokens in JSF

4.4 Security Descriptors Fallacy - Configuration

The declaration of security parameters in web.xml are imperative especially the security elements that are used for preserving the confidentiality and integrity of the ViewState. It has been noticed that declaration of *"ALGORTIHM"* in uppercase in *"org.apache.myfaces.ALGORITHM"* does not initialize the Initialization Vector (IV) in Apache MyFaces. This is a bad design practice and could have devastative impacts on the security of a JSF application. The source code of the *"utils.StateUItils"* class (which holds security configuration elements) as presented in listing 4 which clearly reflects that these parameters have to be applied in lower case but the documentation of various JSF versions is not written appropriately and is not inline with the real code. In other words, the documentation is misleading.

```
public static final String INIT_PREFIX = "org.apache.myfaces.";
public static final String INIT_ALGORITHM = INIT_PREFIX + "
    ALGORITHM";
private static String findAlgorithm(ExternalContext ctx) {

String algorithm = ctx.getInitParameter(INIT_ALGORITHM);
        if (algorithm == null)
        {
            algorithm = ctx.getInitParameter(INIT_ALGORITHM.
                toLowerCase());
        }
 return findAlgorithm( algorithm );
    }

   .. Truncated     ..
```

Listing 4: Explicit Specification - Implementing to Lower Case

4.4.1 Secure Way of Configuring Security Descriptors

The best practice is to declare the configuration parameters in web.xml as presented in listing 5.

```
 <context-param>
     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
     <param-value>client</param-value>
</context-param>

<context-param>
            <param-name>org.apache.myfaces.secret</param-name>
            <param-value>MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIz</param-
                value>
</context-param>

<context-param>
     <param-name>org.apache.myfaces.algorithm</param-name>
     <param-value>AES</param-value>
</context-param>

<context-param>
     <param-name>org.apache.myfaces.algorithm.parameters</param-name
         >
     <param-value>CBC/PKCS5Padding</param-value>
</context-param>

<context-param>
     <param-name>org.apache.myfaces.algorithm.iv</param-name>
     <param-value>NzY1NDMyMTA3NjU0MzIxMA==</param-value>
</context-param>

 <context-param>
  <param-name>org.apache.myfaces.secret.cache</param-name>
            <param-value>false</param-value>
 </context-param>

</web-app>
```

Listing 5: Secure Way of Declaring Encryption Parameters in JSF

This point should be taken into account while doing penetration testing so that enhanced attacks can be tested against the inappropriate implementation of JSF security.

NOTE: It has been noticed that JSF framework only encrypts ViewState in order to provide confidentiality but there is no standard implementation of MAC by default so that the integrity of ViewState is preserved. While we know that JSF security greatly depends on the web.xml file, the MAC functionality is also introduced in it. The developer has to explicitly specify the MAC algorithm, key and caching.

The parameters that are used nowadays are *"org.apache.myfaces.MAC_ALGORITHM"*, *"org.apache.myfaces.MAC_SECRET"* and *" org.apache.myfaces.MAC_SECRET.CACHE"* respectively. Always declare all of these parameters in lower case.

4.5 JSF Version Tracking and Disclosure

JSF configuration has inbuilt configuration parameters that are used to disclose the version of the installed framework. However, it is always taken lightly and is not fixed by the developers or administrators to avoid leakage of the information in HTTP response headers. It has been noticed that version disclosure may result in detecting critical flaws in JSF applications because publicly available exploit databases can be used to fingerprint the security vulnerabilities present in the installed JSF framework.

In Suns RI JSF implementation, the *"com.sun.faces.disableVersionTracking"* configuration parameter is defined explicitly. By default, it is set to false which means application running on the web server will throw the JSF version into the response headers when the web client queries for it. The collaborative disclosure of web server version and framework version can be devastative from the developers point of view but it is fruitful for pen testing purposes. JSF is not immune to security vulnerabilities [12, 13, and 14] as seen in the recent past. These security issues should be taken into consideration while deploying JSF applications because information leakage is the basis of a number of web attacks. We conducted generic tests on several websites that are using JSF and found that more than 85% of the websites are throwing JSF version number in their response headers out of which several of them were running old and vulnerable versions.

4.6 JSF Data Validation

Anyone involved in security understands the importance of proper input validation. JSF offers a few different techniques for validation in order to prevent web attacks such as Cross Site Scripting (XSS). Some of the input validation modules have been available since JSF 1.2, and others are unique to JSF 2.0.

4.6.1 JSF 1.2 Validation

Apache MyFaces and the Apache Tomahawk library provide JSF components that can allow for data validation within the UI page itself. One of the more powerful ways of validating input is by leveraging regular expressions via the ¡t:validateRegExpr¿ tag provided by Tomahawk [15]. Consider an example where we wish to validate a ZIP code. A MyFaces example of data validation using Tomahawk [16] is presented in listing 6.

```
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>

<h:outputLabel for="zip1" value="Zip"/>
<t:inputText value="#{order.zipCode}" id="zip1">
  <t:validateRegExpr pattern="\d{5}" message="ZIP Code"/>
</t:inputText>
```

Listing 6: Generic MyFaces Example - Tomahawk

In this particular example, a regular expression is being used to limit the zip code to five digits. Notice that you can include an error message as well, and all of this is done within the .xhtml page itself. A similar example using Facelets [17] is presented in listing 7.

```
<html ... xmlns:ui="http://java.sun.com/jsf/facelets" xmlns:t="http
    ://myfaces.apache.org/tomahawk">

<h:inputText type="text" id="val
  value="#{SimpleBean.val}" required="true">
   <t:validateRegExpr pattern="[a-zA-Z]{1,100}" />
</h:inputText>
```

Listing 7: Generic Facelets Example

The JSF Reference Implementation (RI), codenamed "Mojarra", comes with its own tag library that also leverages regular expressions. Mojarra's *<mj: regexValidator>* will perform the same operation as discussed above. Furthermore, Mojarra's tag library is armed with an *<mj: creditCardValidator>* to validate the proper format of credit cards [18].

4.6.2 JSF 2.0 Validation

JSF 2.0 contains a collection of tags called validators. These are built in to the JSF 2.0 core library. JSF developers will find the following tags particularly useful for data validation:

- *<f:validateLength>* : use this to validate that input falls between a minimum and maximum length
- *<f:validateLongRange>* : use this to validate that numeric input falls between a minimum and maximum value
- *<f:validateDoubleRange>* : similar to validateLongRange, but used for double values
- *<f:validateRegex>* : use this to leverage regular expression validation

Here is an example of JSF validators in use as presented in listing 8.

```
<tr>
  <td>User ID: <td>User ID:
  <h:inputText value="#{bidBean2.userID}" required="true"
      requiredMessage="You must enter a user ID"
   validatorMessage="ID must be 5 or 6 chars" id="userID">

  <f:validateLength minimum="5" maximum="6"/>
  </h:inputText></td>

  <td><h:message for="userID" styleClass="error"/></td>
</tr>
```

Listing 8: Generic Usage of JSF Validators

Notice in the above example that the *<h: inputText>* contains a required, *requiredMessage* and *validatorMessage* attribute. The required attribute indicates that this value must be input by the end user and an error message is displayed if it is not provided.

4.6.3 Custom Validations

All of the above approaches work well when the values are not tied closely to business logic, or if these validators and tag libraries are suitable to perform the validation we need. Sometimes there is a need to build custom validation components for data types that arent supported by standard JSF validators [19].

In this scenario, the validator attribute of the *<h: inputText>* tag references a validator method that is defined within the bean class [20] as presented in listing 9.

```
 <tr>
<td>Bid Amount: <td>Bid Amount: $<h:inputText value="#{bidBean2.
    bidAmount}" required="true"
requiredMessage="You must enter an amount
converterMessage="Amount must be a number"
validator="#{bidBean2.validateBidAmount}"
id="amount"/>

</td>
  <td><h:message for="amount" styleClass="error"/></td>
</tr>
```

Listing 9: Example : Bid Bean Class

In the BidBean2 class, we would define our custom validation method as presented in listing 10, validateBidAmount():

```
public void validateBidAmount(FacesContext context, UIComponent
    componentToValidate,
UIComponent componentToValidate, Object value) throws
    ValidatorException {

double bidAmount = ((Double)value).doubleValue();
double previousHighestBid = currentHighestBid();

if (bidAmount <= previousHighestBid)
  {
  FacesMessage message =
                          new FacesMessage("Bid must be higher than
                              current "
                         + new FacesMessage(Bid must be higher than
                             current + "highest bid ($"
                         + previousHighestBid + ").");
                         throw new ValidatorException(message);
             }
}
```

Listing 10: Custom Bean Validator Class

As we can see, this approach allows more flexibility while validating input data. The validators play a significant role in curing many security vulnerabilities at the source level.

5 Conclusion

In this paper, we have presented the security issues in JSF architecture. Web frameworks have unique semantics and security models. A thorough understanding of the internal architecture of the frameworks is imperative to undertake productive penetration testing. We have discussed the advance features that should be tested for complete and extensive penetration testing of JSF. In general, JSF is similar to ASP.NET from security perspective but differs in deployment of inherent security controls. Every web framework is required to be dissected so that step by step testing can be performed in a robust manner.

6 About the Authors

Aditya K Sood is a Senior Security Practitioner and PhD candidate at Michigan State University. He has already worked in the security domain for Armorize, COSEINC and KPMG. He is also a founder of SecNiche Security Labs, an independent security research arena for cutting edge computer security research. He has been an active speaker at industry conferences and already spoken at RSA, HackInTheBox, ToorCon, HackerHalted, Source, TRISC, AAVAR, EuSecwest, XCON, Troopers, OWASP AppSec USA, FOSS, CERT-IN, etc. He has written content for Virus Bulletin, HITB Ezine, Hakin9, ISSA, ISACA, CrossTalk, Usenix Login, and Elsevier Journals such as NESE and CFS. He is also a co author for debugged magazine.

He is also associated with Cigital Labs for doing applied security research in the field of software and application security.

Email: adi_ks [at] secniche.org

Personal Website: http://www.secniche.org
Company Website : http://www.cigital.com

Krishna Raja is a Senior Application Security Consultant at Security Compass with an extensive background in Java EE application development. He has performed comprehensive security assessments for financial, government, and health care organizations across Canada and the United States. Krishna has carried out the role of security advisor, security analyst, project manager and trainer. He has given lectures and taught courses at RSA, Source Boston, ISSA Secure San Diego and OWASP AppSec DC. Krishna graduated from the University of Western Ontario in 2004 with an Honors BSc. in Computer Science with Software Engineering specialization. He is also an ISC2 Certified Secure Software Lifecycle Professional (CSSLP).

Email: krish [at] securitycompass.com
Company Website: http://www.securitycompass.com

7 References

[1] Apache Java Server Faces, http://myfaces.apache.org/

[2] ViewState Decoder, http://www.pluralsight-training.net/community/media/p/51688.aspx

[3] POET Tool, http://netifera.com/download/poet/poet-1.0.0-win32-x86.jar

[4] Deface Tool, https://github.com/SpiderLabs/deface

[5] Beware of Serialized GUI Objects Bearing Data, http://www.blackhat.com/presentations/bh-dc-10/Byrne_David/BlackHat-DC-2010-Byrne-SGUI-slides.pdf

[6] Padding Oracle Attacks, http://www.usenix.org/event/woot10/tech/full_papers/Rizzo.pdf

[7] Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET, http://www.ieee-security.org/TC/SP2011/PAPERS/2011/paper030.pdf

[8] Automated Padding Oracle Attacks with PadBuster, http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-with-padbuster/

[9] Cracking ViewState Encryption in JSF, http://www.youtube.com/watch?v=euujmKDxmC4

[10] Microsoft Patch Padding Oracle Attacks, http://www.microsoft.com/technet/security/bulletin/ms10-070.mspx

[11] Using Padding in Encryption, http://www.di-mgt.com.au/cryptopad.html

[12] Sun Java Server Faces Cross-Site Scripting Vulnerability, http://www.securityfocus.com/bid/28192

[13] Apache MyFaces Tomahawk JSF Framework Cross-Site Scripting (XSS) Vulnerability , http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=544

[14] Sun Glassfish Enterprise Server - Multiple Linked XSS vulnerabilies, http://dsecrg.com/pages/vul/show.php?id=134

[15] MyFaces Tomahawk, http://myfaces.apache.org/tomahawk/index.html

[16] Tomahawk ValidateRegExpr, http://www.developersbook.com/jsf/myfaces/tomahawk-tag-reference/tomahawk-validateRegExpr.php#1

[17] Facelets, http://facelets.java.net/

[18] JSF Majorra Extension Tags Validation and Focus, http://java.sys-con.com/node/1031733

[19] JSF Validation Tutorial, http://www.mastertheboss.com/web-interfaces/293-jsf-validation-tutorial.html?showall=1

[20] JSF: Validating User Input, http://courses.coreservlets.com/Course-Materials/pdf/jsf/08-Validation.pdf